Reporte de algoritmos de ordenamiento

José Manuel Tapia Avitia.
Matrícula: 1729372
jose.tapiaav@gmail.com
https://github.com/jose-tapia/1729372MC

1 de septiembre de 2017

El presente reporte tiene la finalidad de mostrar a grandes rasgos las caracteristicas principales y el funcionamiento de algunos algoritmos de ordenamiento.

1. Selection Sort

El algoritmo de selección es el que a mi consideración es el más intuitivo.
Al inicio, busca el menor elemento (de acuerdo a nuestra función de comparación) y lo intercambia por el elemento en la primera posición. Después, busca el menor elemento que se encuentre a partir de la segunda posición, intercambiandolo con la segunda posición.

El algoritmo se repite hasta llegar a la última posición, en donde se tendría el arreglo ordenado.

```
1  def selectionSort(arr):
2      Para i igual a 0, 1, 2, ..., len(arr) − 1:
3          valorMinimo = arr[i];
4          posicion = i;
5          Para j igual a i, i + 1, ..., len(arr) − 1:
6              Si arr[j] < valorMinimo:
7                  valorMinimo = arr[j];
8                  posicion = j;
9          Intercambiar el valor de arr[i] y arr[posicion];
10     Regresar arr
```

La cantidad de operaciones que se realizan tiende a ser cuadrática con respecto al tamaño del arreglo.

Sea n el tamaño del arreglo, en la primera iteración del bucle, para buscar el mínimo realiza n comparaciones (ya que inclusive se compara consigo mismo), en la segunda iteración del bucle, realiza $n-1$ comparaciones, en la tercera iteración, realizaría $n-2$, ... en la i-ésima iteración realizaría $n-i+1$ comparaciones para buscar el mínimo, es decir, que el número total de comparaciones que se realizaron fueron

$$n + (n-1) + (n-2) + \cdots + (n-i+1) + \cdots + 1 = \frac{n(n+1)}{2}$$

Por lo tanto es algo lento el algoritmo para arreglos de considerable tamaño.
Complejidad tiempo: $\mathcal{O}(n^2)$

2. Bubble Sort

El siguiente algoritmo tiene similitud al comportamiento de las burbujas en un vaso de refresco por ejemplo, mientras suben las burbujas de gas, estás se van haciendo más grandes, algo parecido sucede en el algoritmo.

Denominaremos el siguiente proceso como *burbujear* : Compararemos el primer elemento y el segundo, si el primero es mayor al segundo, los intercambiaremos, si no, no. Independientemente, compararemos el segundo y tercer elemento, si el segundo es mayor al tercero, los intercambiaremos, si no, no. Compararemos el tercer y cuarto elemento, si el tercero es mayor al cuarto, los intercambiaremos, si no, no. Así hasta llegar a comparar el penúltimo y último elemento, si el penúltimo es mayor al ultimo, los intercambiaremos, si no, no.

El algoritmo consiste en *burbujear* hasta que el arreglo este ordenado. Podemos notar que en el primer *burbujear* la última posición queda el mayor elemento de la lista. Al realizar un segundo *burbujear*, tenemos que en la penúltima posición queda el segundo mayor elemento de la lista.

Al realizar una cantidad de *burbujear* equivalente al tamaño del arreglo se aseguraría que el arreglo este ordenado.

```
1 def bubbleSort(arr):
2 |   Para i igual a 0, 1, 2, ..., len(arr) − 1:
3 |   |   Para j igual a 0, 1, ..., len(arr) − i − 2:
4 |   |   |   Si arr[j + 1] < arr[j]:
5 |   |   |   |   Intercambiar el valor de arr[j] y arr[j + 1];
6 |   Regresar arr
```

Sea n el tamaño del arreglo. El i-ésimo *burbujear* tarda $n - i - 1$ comparaciones (No comparamos los últimos i elementos puesto que ya sabemos que están ordenados, ahorrando unas cuantas operaciones). Entonces, en total se estarían realizando

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{(n-1)n}{2}$$

Por lo tanto se tarda cuadrático en función del tamaño del arreglo.

Complejidad tiempo: $\mathcal{O}(n^2)$

3. Insertion Sort

El siguiente algoritmo puede que ya lo hayamos hecho de manera involuntaria, describiremos el algoritmo con una similitud a la inducción matemática:

Consideremos el arreglo formado por el primer elemento solamente, por ser el único elemento, el arreglo ya esta ordenado.

Supongamos que hasta una cierta i el arreglo formado con los primeros i elementos esta ordenado. Para ordenarlo para los primeros $i + 1$ elementos, tomamos el elemento $i + 1$ y checamos si es mayor al elemento i, si lo es, intercambiamos los valores, en caso contrario ya tendríamos el arreglo ordenado. Si intercambiamos los valores, existe la posibilidad de que el elemento $i - 1$ sea mayor que el elemento que estamos agregando, si lo es, los intercambiamos, en caso contrario, tendríamos el arreglo ordenado.

Siguiendo este procedimiento tendíamos que el elemento que estamos agregando quedaría en su posición respectiva, teniendo ordenados los primeros $i+1$ elementos del arreglo. Cómo el arreglo formado con el primer elemento ya esta ordenado, podemos ordenar con los primeros dos, luego los primeros tres, después los primeros cuatro, así hasta tener ordenado el arreglo completo.

```
1 def insertionSort(arr):
2 |   Para i igual a 0, 1, 2, ..., len(arr) − 1:
3 |   |   Para j igual a i − 1, i − 2, ..., 0:
4 |   |   |   Si arr[j + 1] < arr[j]:
5 |   |   |   |   Intercambiar el valor de arr[j] y arr[j + 1];
6 |   |   |   en caso contrario:
7 |   |   |   |   Romper el ciclo;
8 |   Regresar arr
```

Este algoritmo es más eficiente que los anteriores puesto a que en el mejor de los casos, no intercambia en ningún momento, es decir, que solo se tardaría en recorrer el arreglo. En el peor de los casos realizaría todos los intercambios posibles, es decir, en la i-ésima iteración se pueden realizar a lo más $i-1$ intercambios (que sería que el elemento i es el menor de la lista y debe ser desplazado hasta la primera posición), en el peor de los casos se realizarían

$$0+1+2+\cdots+(n-2)+(n-1)=\frac{(n-1)n}{2}$$

Lo cual es cuádratico en función del tamaño del arreglo.

Complejidad tiempo en el mejor de los casos $\mathcal{O}(n)$, en el peor de los casos $\mathcal{O}(n^2)$.

4. Quick Sort

El siguiente algoritmo es el más óptimo, ya que su promedio de complejidad tiempo es $\mathcal{O}(nlogn)$, lo cuál es mucho mejor que el $\mathcal{O}(n^2)$ de los algoritmos pasados.

El algoritmo consiste en:

Dado un arreglo que se desea ordenar, se toma un elemento del mismo, el cual le llamaremos *pivote*. En caso de que sea un arreglo vacío, tendríamos que ya esta ordenado, es decir, si no tiene elementos, regresaremos el arreglo vacío.

El pivote puede ser cualquier elemento, tanto el primero como el último, se puede tomar de manera aleatoria, pero para ser más prácticos, tomaremos el pivote como el primer elemento del arreglo.

Crearemos dos arreglos, los cuales llamaremos izq y der, en donde en el arreglo izq tendremos a todos los elementos del arreglo menores a *pivote* y en der tendremos todos los elementos del arreglo mayores o iguales a *pivote* exceptuando a nuestro pivote, se puede apreciar que la unión de izq, der y *pivote* forman el arreglo original.

Si ordenamos el arreglo izq y der con el mismo algoritmo de manera recursiva, es decir, realizando el mismo procedimiento, tendríamos que el arreglo ordenado que nosotros buscamos (El arreglo inicial) sería la concatenación de el arreglo ordenado izq, el arreglo formado por el elemento *pivote* y el arreglo ordenado der en ese orden. Los concatenamos y regresamos tal arreglo.

```
1  def quickSort(arr):
2      Si arr es un arreglo vacío:
3          Regresar arr
4      Crear arreglos vacíos izq y der;
5      pivote = arr[0] ;
6      Para  i igual a 1, 2, ..., len(arr) − 1:
7          Si arr[i] <pivote:
8              agregar arr[i] a izq;
9          en caso contrario:
10             agregar arr[i] a der;
11     Regresar Concatenación de los arreglos quickSort (izq), [pivote], quickSort
         (der)
```

El algoritmo es relativamente sencillo de programar, teniendo en cuenta la ventaja de su tiempo promedio mejor que los demás, vale la pena gastar un poco más de tiempo para programarlo. Algunas dudas que se podrían tener es el por que se retorna el arreglo vacío. Hay casos especiales en los que es conveniente esta consideración, mencionaremos algunos de ellos; en los que el arreglo sea el único elemento, se tendría que izq y der estarían vacios, al nosotros retornarlos tal cual, no afectaría en nada la concatenación, retornando el arreglo bien ordenado. El caso en el que el pivote sea el menor elemento (o el mayor) del arreglo, se tendría que el arreglo izq (o der) estarían vacíos, lo cual al ordenar el der (o el izq) y concatenarlo con [$pivote$] y el otro arreglo vacío no habría ningún problema, es decir, igual retornaría el arreglo ordenado.

De hecho, en el caso en el que siempre el *pivote* este en los extremos, se tendría el peor de los casos, consiguiento una complejidad de $\mathcal{O}(n^2)$, por eso se recomienda tomar el *pivote* de manera aletoria, aumentando las probabilidades de conseguir una mejor complejidad.

Complejidad tiempo: promedio $\mathcal{O}(nlogn)$, en el peor de los casos $\mathcal{O}(n^2)$.